How Design Predicts Failures

Adrian Schröter Thomas Zimmermann Andreas Zeller
Saarland University, Department of Computer Science, Saarbrücken, Germany
{adrian,zimmerth,zeller}@st.cs.uni-sb.de

Abstract

In an empirical study of 52 ECLIPSE plug-ins, we found that the software design as well as past failure history, can be used to build support vector machines which accurately predict failure-prone components in new programs. Our prediction only requires usage relationships between components, which are typically defined in the design phase; thus, designers can easily explore and assess design alternatives in terms of predicted quality.

1 Introduction

In this work, we aim to *predict* how failure-prone a component will be by learning from history which design decisions correlated with failures in the past. More precisely, in contrast to research that used metrics to predict failure-proneness [2, 1, 3], we focus on *usage relationships:* Does using the library A increase or decrease the risk of failure? It turns out that this information alone suffices to predict the failure-proneness of individual components.

2 Failure-Proneness of Single Components

Why does usage influence the risk of failure? In a first experiment, we checked our hypothesis that the usage of certain components such as packages or classes correlates with failures. For every ECLIPSE component C, we computed the likelihood $p(\text{✘}|C)$ that a post-release failure occurred when this component was used in a file:

$$p(\text{✘}|C) = \frac{\text{number of files using } C \text{ with failures}}{\text{number of files using } C} = \frac{N^C_{\text{✘}}}{N^C_{\text{all}}}$$

We tested for every component with two *t-tests* ($\alpha = 0.05$) whether its likelihood is significantly higher (or lower) than the average likelihood $p(\text{✘}) = 1649/6751 = 0.244$ of a failure and the likelihood $p(\text{✘}|\neg C)$ of a failure when *not* using component C.

As an example, consider two packages from the ECLIPSE API. The package *org.eclipse.jdt.core.compiler* provides an interface to the JAVA compiler, such as invoking it or accessing compilation results. On the other hand, the *org.eclipse.ui* package gives access to the ECLIPSE user interface, providing GUI elements for user interaction. For the average programmer, dealing with user interfaces is an everyday's job—in contrast to interacting with a compiler package. Therefore, we would assume that code which uses the *compiler* package is more error-prone than code that uses the *ui* package. But can we actually quantify these differences?

Package C	N^C_{all}	$N^C_{\text{✘}}$	$p(\text{✘}\|C)$
org.eclipse.jdt.internal.compiler.lookup.*	197	170	0.8629
org.eclipse.jdt.internal.compiler.*	138	119	0.8623
org.eclipse.jdt.internal.compiler.ast.*	132	111	0.8409
org.eclipse.jdt.internal.compiler.util.*	148	121	0.8175
org.eclipse.jdt.internal.ui.preferences.*	63	48	0.7619
org.eclipse.jdt.core.compiler.*	106	76	0.7169
...			
org.eclipse.swt.custom.*	233	41	0.1760
org.eclipse.pde.internal.ui.*	211	35	0.1659
org.eclipse.jface.resource.*	387	64	0.1654
org.eclipse.pde.core.*	112	18	0.1608
org.eclipse.jface.wizard.*	230	36	0.1566
org.eclipse.ui.*	948	141	0.1488

Table 1: Good and bad imports (packages) in ECLIPSE

To this end, we have mined the ECLIPSE bug and version archives for usage data (which component uses which other components?) as well as for failure rates (which component had how many failures in the past?). Applied to the *compiler* and *ui* packages above, we found that 71% of the components using the *compiler* package needed to be fixed due to a failure. However, only 14% of the components using *ui* had to be fixed. Hence, we can confirm: compiler-related code is more failure-prone than GUI-related code.

With such a prediction, a software designer can explore and assess the design alternatives and check which one has the lowest risk. She can also use the prediction to identify the components that are most likely to fail afterwards, and assign appropriate quality assurance efforts—and all this at design time, when decisions matter most.

In ECLIPSE, using the compiler and the GUI are just two extremes of how the usage of individual components impacts later failures. In this work, we therefore investigate whether *combinations* of usages make good failure predictors: If a component uses, say, the GUI and version control, but not the compiler, this specific *usage pattern* may indeed correlate with failures—and this is what we want to mine, as a model that makes accurate failure predictions.

3 Building Prediction Models

We built prediction models from import relationships using *support vector machines* with a radial basis function as kernel. We used the imported components of a file as input features to predict the number of post-release failuires. All models were trained with historic design data. Such models help us to address two problems:

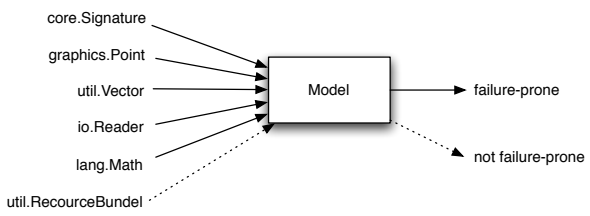

Figure 1: Classification of a component

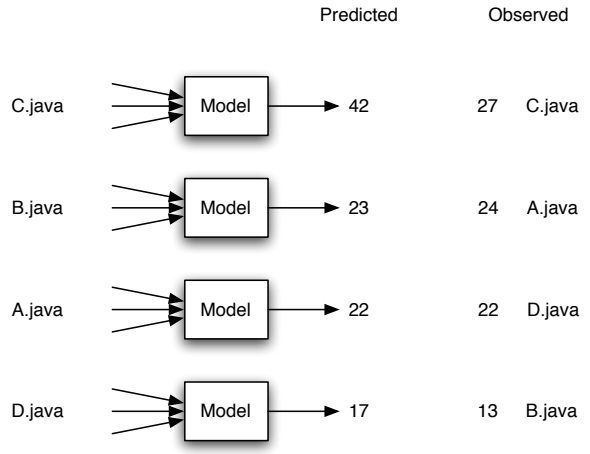

Figure 2: Comparison of predicted and observed ranking

Classification. Can we tell whether a component will be failure-prone based on its design? This helps to mark risky parts of a software design.

Ranking. Can we tell which components will have the most failures? This information identifies the parts of a software design that require most attention when being implemented and tested.

Figure 1 illustrates the classification of files as failure-prone. If a file imports five classes *core.Signature*, *graphics.Point*, *util.Vector*, *io.Reader*, and *lang.Math*—it is classified as failure-prone by our model. If another file additionally imports *util.ResourceBundle*, our model classifies it as not failure-prone. For ranking we predict for every component the number of failures and sort by this. We compare predicted rankings—such as *C.java*, *B.java*, *A.java*, and *D.java* in Figure 2—to the observed rankings.

4 Results: 52 ECLIPSE Plug-Ins

For our experiments, we used *random splits*: we randomly chose one third of the 52 plug-ins of ECLIPSE version 2.0 as our training set, the other two third as test set for 2.0 and the complement in 2.1 as test set for 2.1. We generated a total of 40 random splits; we averaged the results for computing the *precision*, *recall*, and *correlation* values.

Precision and Recall. For the test sets in version 2.0, our models obtained a *precision* of 0.6671 (see Table 2). That is, two out of three components predicted as failure-prone were observed to produce failures. For a random guess instead, the precision is only 0.365.

	Precision	Recall	Spear. Coef.
training	0.8770	0.8933	0.5961
test in v2.0	0.6671	0.6940	0.3002
→5%	0.7861		0.1369
→10%	0.7875		0.2032
→15%	0.7957		0.2648
→20%	0.8000		0.3190
test in v2.1	0.5917	0.7205	0.2842
→5%	0.8958		0.3416
→10%	0.8399		0.3702
→15%	0.7784		0.3675
→20%	0.7668		0.3615

Table 2: Predicting packages with classes in ECLIPSE

The *recall* of 0.6940 for the tests in version 2.0 (see Table 2) indicates that two third of the observed failure-prone components were actually predicted as failure-prone. Again, a random guess yields only a recall of 0.365.

Ranking vs. Classification. The low values for the Spearman rank *correlation* coefficient in Table 2 indicate that our predicted rankings do not correlate with the observed rankings. However, the precision values for the top 5% are higher than the overall values. This means that our classification works best for the parts that are highly ranked.

Applying Models across Versions. The results for the test set of ECLIPSE version 2.1 show a similar behavior to the ones for version 2.0. This indicates that our model is robust over time, i.e., we can learn a model for one version and apply it to a later version without losing predictive power.

5 Conclusion

A component's likelihood to fail is significantly determined by the set of components that it uses. Why is this so? Our hypothesis is that the set of used components is *determined by the problem domain*and some of these domains are harder to get right than others—at least, this is what the ECLIPSE failure history suggests.

Using our approach, managers and developers can leverage earlier failure history to predict future failure-prone components, and thus assign resources to those components which need it most. Since the set of used components is typically defined at design time, these decisions can be made very early in the software process.

For ongoing information on the project, log on to

`http://www.st.cs.uni-sb.de/softevo/`

References

[1] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Software Eng.*, 22(10):751–761, 1996.

[2] G. Denaro, S. Morasca, and M. Pezzè. Deriving models of software fault-proneness. In *International Conference on Software Engineering and Knowledge Engineering*, pages 361–368, July 2002.

[3] R. Subramanyam and M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Trans. Software Eng.*, 29(4):297–310, 2003.